Evolutionary Programming as a Solution Technique for the Bellman Equation*

Paul Gomme
Federal Reserve Bank of Cleveland, P.O. Box 6387, Cleveland, OH 44101–1387,
Simon Fraser University, Burnaby, B.C., V5A 1S6, CANADA, and
CREFE/UQAM, Case postale 8888, succursale centre-ville, Montréal, Québec, H3C 3P8, CANADA
gomme@sfu.ca

First Draft: April 1996
This Draft: October 1997

Abstract: Evolutionary programming is a stochastic optimization procedure which has proved useful in optimizing difficult functions. It is shown that evolutionary programing can be used to solve the Bellman equation problem with a high degree of accuracy and substantially less CPU time than Bellman equation iteration. Future applications will focus on sometimes binding constraints – a class of problem for which standard solutions techniques are not applicable.

Keywords: evolutionary programming, bellman equation, value function, computational techniques, stochastic optimization

*The financial support of the Social Sciences and Humanities Research Council (Canada) is gratefully acknowledged. The views stated herein are those of the author and are not necessarily those of the Federal Reserve Bank of Cleveland or of the Board of Governors of the Federal Reserve System.

1. Introduction

Stochastic optimization algorithms, like evolutionary programming, genetic algorithms and simulated annealing, have proved useful in solving difficult optimization problems. In this context, a difficult optimization problem might mean: (1) a non-differentiable objective function, (2) many local optima, (3) a large number of parameters, or (4) a large number of configurations of parameters.[1] Thus far, there are few economic applications of such procedures, with most attention has focused on genetic algorithms; see, for example, Arifovic (1995, 1996). This paper explores the potential of evolutionary programming as a solution procedure for solving Bellman equation (value function) problems.

Whereas genetic algorithms include a variety of operators (for example, mutation, cross-over and reproduction), evolutionary programs use *only* mutation. As such, an evolutionary program can be viewed as a special case of a genetic algorithm. The basics of evolutionary programming can be described as follows. Let $X \in \mathbb{R}^n$ be the parameter space and let $x^i \in X$ denote candidate solution $i \in \{1, \ldots, m\}$. If the objective function is $f: X \to \mathbb{R}$, then $f(x^i)$ is the evaluation for element i. Given some initial population, $\{x^i\}_{i=1}^m$, proceed as follows:

(1) Sort the population from best to worst according to the function f.

(2) For the worst half of the population, replace each member with a corresponding member in the top half of the population, adding in some 'random noise.'

(3) Re-evaluate each member according to f.

(4) Repeat until some convergence criterion is satisfied.

The 'noise' added in step (2) helps the evolutionary program to escape local minima and at the same time explore the parameter space. As the amount of noise in step (2) is reduced, the evolutionary program will typically converge to a solution arbitrarily close to the optimum. Properties of evolutionary programs have been explored by a number of authors including Fogel (1992).

There are a number of complications which arise in applying an evolutionary program to the Bellman problem. The most important complication is that the algorithm must solve

[1] A classic example is the *traveling salesman problem* in which a salesman wishes to minimize the distance traveled in visiting a set of N cities.

for the objective function. That is, for the typical evolutionary program, the function f above is *known*. Here, the value function, which depends on the state, is unknown *a priori* and the solution algorithm must solve for the value function—which is also the 'fitness' criterion used to evaluate candidate solutions.

The basics of the algorithm are discussed in Section 2. The specific application is the neoclassical growth model. In the most basic version of the model, the parameters to choose are next period's capital stock (as a function of this period's capital stock). These are restricted to lie in a discrete set. For problems with a large number of capital stock grid points, it is shown that the evolutionary program delivers decision rules arbitrarily close to the known solution, and does so much faster than Bellman equation iteration; see Section 3. Also in Section 3, the performance of the evolutionary program is evaluated when a labor-leisure choice is introduced. For large problems, the evolutionary program is again substantially faster than Bellman equation iteration. Section 4 concludes.

2. The Problem and Algorithm

The specific application is the neoclassical growth model:

$$\max_{\{c_t,\, k_{t+1}\}_{t=0}^{\infty}} \mathrm{E}_0 \left\{ \sum_{t=0}^{\infty} \beta^t \ln c_t \right\}, \qquad 0 < \beta < 1 \tag{1}$$

subject to

$$c_t + k_{t+1} = z_t k_t^{\alpha} + (1-\delta)k_t, \qquad 0 < \delta, \alpha < 1, \qquad t = 0, 1, \ldots \tag{2}$$

where c_t is consumption, k_t is capital, z_t a technology shock, U a well-behaved utility function, and F a well-behaved production function. The associated Bellman equation (value function) is:

$$V(k_t, z_t) \equiv \max_{\{c_t,\, k_{t+1}\}} \{\ln c_t + \beta \mathrm{E}_t V(k_{t+1}, z_{t+1})\} \tag{3}$$

subject to (2). One way to solve this problem is via Bellman equation iteration: given some initial guess $V_0(k_t, z_t)$, iterate on (3) as

$$V_{j+1}(k_t, z_t) \equiv \max_{\{c_t,\, k_{t+1}\}} \{\ln c_t + \beta \mathrm{E}_t V_j(k_{t+1}, z_{t+1})\} \text{ subject to (2)} \tag{4}$$

until either the decision rules converge, or the value function converges. To implement this procedure computationally, the capital stock is restricted to a grid, $\mathcal{K} = \{k^1, k^2, \ldots, k^{NK}\}$.

The technology shock is likewise restricted to $\mathcal{Z} = \{z^1, z^2, \ldots, z^{NZ}\}$. z_t is assumed to follow a Markov chain:

$$\text{prob}\{z_{t+1} = z_j | z_t = z_i\} = \phi_{ij}. \tag{5}$$

When there is 100% depreciation ($\delta = 1$), a closed-form solution can be obtained:

$$k_{t+1} = \alpha\beta z_t k_t^\alpha \tag{6a}$$

$$c_t = (1 - \alpha\beta) z_t k_t^\alpha. \tag{6b}$$

These *known* solutions will be useful in evaluating the performance of the evolutionary program.

The biggest problem with Bellman equation iteration is the *curse of dimensionality*: large capital stock grids or additional endogenous state variables make the maximization in (4) computationally expensive. In many ways, the problem as set out in (4) looks like a natural application for an evolutionary program: for each of the $NK \times NZ$ grid points in the state space, there are NK potential values for k_{t+1}. While $V_j(k_t, z_t)$ is known at iteration j, the limiting value function,

$$V(k_t, z_t) \equiv \lim_{j \to \infty} V_j(k_t, z_t) \tag{7}$$

is generally unknown. If $V(k_t, z_t)$ were known, this would be a straightforward evolutionary program application. However, the algorithm must also iterate on $V_j(k_t, z_t)$ to obtain an approximation to $V(k_t, z_t)$. It is this iteration which distinguishes the neoclassical growth model from the typical evolutionary program application.

At each iteration in (4), there is a solution for next period's capital stock,

$$k_{t+1} = K_j(k_t, z_t) \in \mathcal{K}. \tag{8}$$

Rather than obtain this by maximization, suppose one were to 'guess' a set of solutions,

$$k_{t+1} = K^i(k_t, z_t) \in \mathcal{K}, \qquad i \in \{1, 2, \ldots, m\}. \tag{9}$$

For each $i \in \{1, 2, \ldots, m\}$ can be computed

$$V^i(k_t, z_t) = \ln c_t + \beta \mathrm{E}_t V_j(K^i(k_t, z_t), z_{t+1}) \tag{10}$$

where

$$c_t = z_t k_t^{\alpha} + (1-\delta)k_t - K^i(k_t, z_t). \tag{11}$$

For each i, this results in $NK \times NZ$ numbers (one for each of the grid points for the state space). So that each guess has as scalar value associated with it, compute

$$\overline{V}^i = \frac{1}{NK \times NZ} \sum_{k_t \in \mathcal{K}} \sum_{z_t \in \mathcal{Z}} V^i(k_t, z_t). \tag{12}$$

Next, sort the guesses such that

$$\overline{V}^1 > \overline{V}^2 > \cdots > \overline{V}^m. \tag{13}$$

At the next iteration, elements $i \in \{m/2 + 1, \ldots, m\}$ will be replaced as follows:

$$K^i(k_t, z_t) = k^p \in \mathcal{K} \tag{14}$$

where

$$p = \max[\min[q + \text{INT}(x), NK], 1], \tag{15}$$

q is the index to the capital stock grid point corresponding to $K^{i-m/2}(k_t, z_t)$, INT takes the integer portion of of a real number, and x is a random number drawn from $N(0, \sigma^2)$. The procedure in (14) is repeated for each $k_t \in \mathcal{K}$ and for each $z_t \in \mathcal{Z}$. A new random number x is drawn for each grid point. The upshot of this procedure is to replace the worst half of the population of guesses with the best half, plus some noise.

How should $V_j(k_t, z_t)$ be updated for the next iteration? In the spirit of the maximization in (4), let

$$V_{j+1}(k_t, z_t) = \max_{i \in \{1, \ldots, m\}} [V^i(k_t, z_t)], \text{ for each } k_t \in \mathcal{K} \text{ and } z_t \in \mathcal{Z}. \tag{16}$$

Another alternative would have been to have set $V_{j+1}(k_t, z_t) = V^1(k_t, z_t)$ (the value function for the best guess). As a practical matter, the maximization in (16) speeds convergence.

In experimenting with the algorithm, it was prudent to replace guess $K^{m/2}(k_t, z_t)$ with the rule which implements the maximum in (16). Since this replaces the worst guess in the top half of the population, it does not overwrite a particularly good guess. Further, if the replacement is a bad thing to do, the value associated with this rule will presumably place

it in the bottom half of the population next iteration, and it will be discarded. Intuitively, this is like performing the maximization associated with Bellman equation iteration, but checking only a small subset of the possible values for next period's capital stock. Again, as a practical matter, this replacement greatly speeds convergence.

To finish this section, the evolutionary program will be summarized.

(1) Generate an initial guess for the value function, $V_0(k_t, z_t)$, and a population of candidate solutions, $\{K^i(k_t, z_t)\}_{i=1}^m$ for $k_t \in \mathcal{K}$ and $z_t \in \mathcal{Z}$. Also, set an initial value for σ which governs the amount of 'noise' introduced to decision rules when they are copied.

(2) For each rule $i \in \{1, 2, \ldots, m\}$, compute $V^i(k_t, z_t)$ via (10) and (11), and compute $\overline{V}^i$ using (12).

(3) Sort the population as in (13).

(4) Compute $V_{j+1}(k_t, z_t)$ using (16). Replace rule $^{\mathrm{m}}/_2$ with that which would achieve this maximum.

(5) Replace the bottom half of the population with perturbed members of the top half of the population as described in (14).

(6) Repeat (2)–(5) until converge is achieved, or a prespecified number of iterations are completed.

(7) Reduce σ (the amount of experimentation).

(8) Repeat (2)–(7) until σ is sufficiently small.

3. Calibration and Results

In this section, the evolutionary program is compared to Bellman equation iteration both in terms of accuracy and computational requirements. Two major cases are considered: with and without a labor-leisure choice. Subcases are presented for closed-form vs. nonclosed-form, and stochastic vs. nonstochastic technology shocks (z_t).

3.1. No Labor–Leisure Choice

Table 1 presents parameter values common to all experiments in this section. For the most part, these are values typically used in the real business cycle literature; see, for example, Prescott (1986). The capital stock grid was specified as a set of evenly spaced points on the interval $[\underline{k}, \overline{k}]$; the upper and lower bounds on the capital stock were chosen such that the ergodic set for capital was strictly contained in $[\underline{k}, \overline{k}]$. The set for the technology shock was specified as having two points:

$$\mathcal{Z} = \{\underline{z}, \overline{z}\}.$$

The technology shock evolves as:

$$\text{prob}[z_{t+1} = \underline{z} | z_t = \underline{z}] = \text{prob}[z_{t+1} = \overline{z} | z_t = \overline{z}] = \pi.$$

The transition probability, π, and values for $\underline{z}$ and $\overline{z}$ were chosen to match the properties of Solow residuals as reported in Prescott (1986).

Parameter	Description	Value
α	capital's share of income	0.36
β	discount factor	0.99
$\underline{k}$	lower bound for capital grid	$1/4 \times$ steady state
$\overline{k}$	upper bound for capital grid	$2 \times$ steady state
$\underline{z}$	lower bound for technology shock	$e^{-0.00763}$
$\overline{z}$	upper bound for technology shock	$e^{0.00763}$
π	persistence of technology shock	0.975

Table 1: Parameter values used in computational exercises.

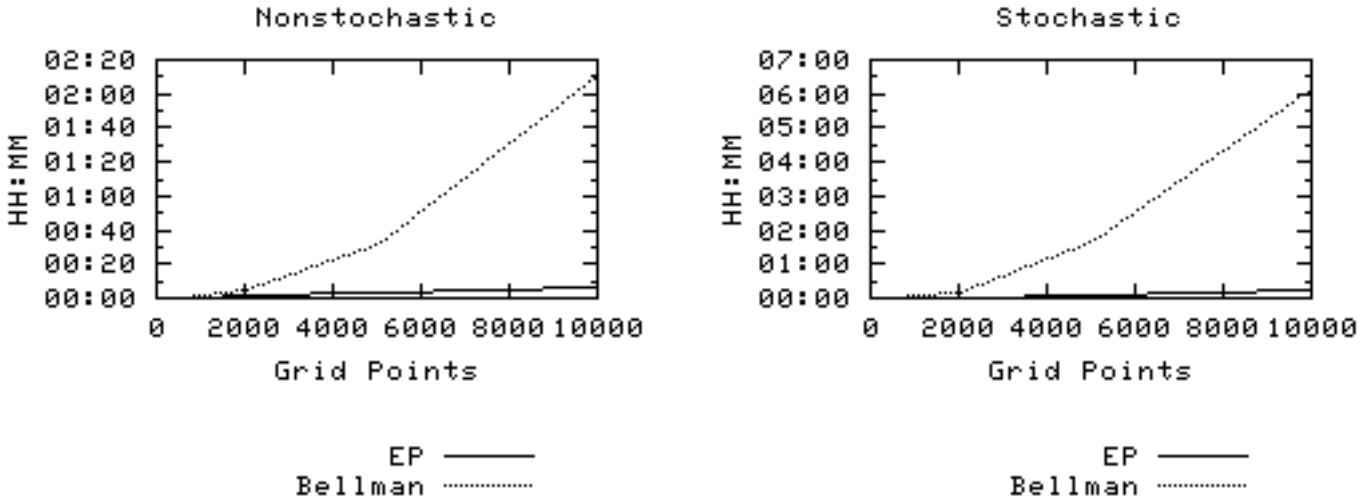
In terms of initial conditions,

$$V_0(k_t, z_t) = 0 \qquad \forall k_t,\ \forall z_t, \tag{17}$$

and

$$K^i(k_t, z_t) = \underline{k} \qquad \forall k_t,\ \forall z_t,\ \forall i. \tag{18}$$

Figure 1: CPU time for closed-form case.



(18) ensures that consumption is always positive for the initial guesses.[2] σ, which governs the amount of experimentation in the evolutionary program, starts at $NK/10$. Its value is halved at each step (7) (see the end of Section 2) until its value is less than 0.1. Iterations leading to step (7) continue until there has been no change in the decision rule generating the best solution for 20 iterations, or until a total of 50 iterations have been completed.

Table 2: Results for the closed-form case: $\delta = 1$.

	Nonstochastic		Stochastic	
Grid Points	Evolutionary Program	Bellman Iteration	Evolutionary Program	Bellman Iteration
100	1.3	0.6	2.7	1.4
200	2.8	2.5	6.0	6.0
500	8.9	16.9	20.8	38.5
1,000	22.1	1:10.6	52.6	2:39.5
2,000	56.7	5:12.1	2:09.4	11:21.5
5,000	2:58.1	31:24.1	6:58.8	1:40:23.9
10,000	6:48.9	2:11:11.3	15:33.9	6:08:11.8

Notes: In all cases, the solutions were within one grid point of the known solutions given in (6*a*) and (6*b*). Reported CPU time is the `user` time reported by the Unix `time` command on a SPARCstation 20 with a 100 MHz HyperSPARC chip.

[2] For the evolutionary program, positive consumption cannot be guaranteed at future stages. When a rule specifies nonpositive consumption, the value function at that grid point evaluates to -10^{10}.

Results for the case in which a closed-form solution is available are reported in Table 2; these results are summarized in Fig. 1. Both the evolutionary program and Bellman equation iteration successfully solved this case in that the final solutions were within one grid point of the known solution. For moderate sized grids (up to 200 grid points for capital), Bellman equation iteration is actually faster than the evolutionary program. This ranking is reversed for large grids. For example, with 10,000 grid points, the evolutionary program is more than 20 times faster than Bellman equation iteration. These differences matter: when the technology shock is stochastic, the evolutionary program solves in under 16 minutes while Bellman equation iteration takes over 6 hours.

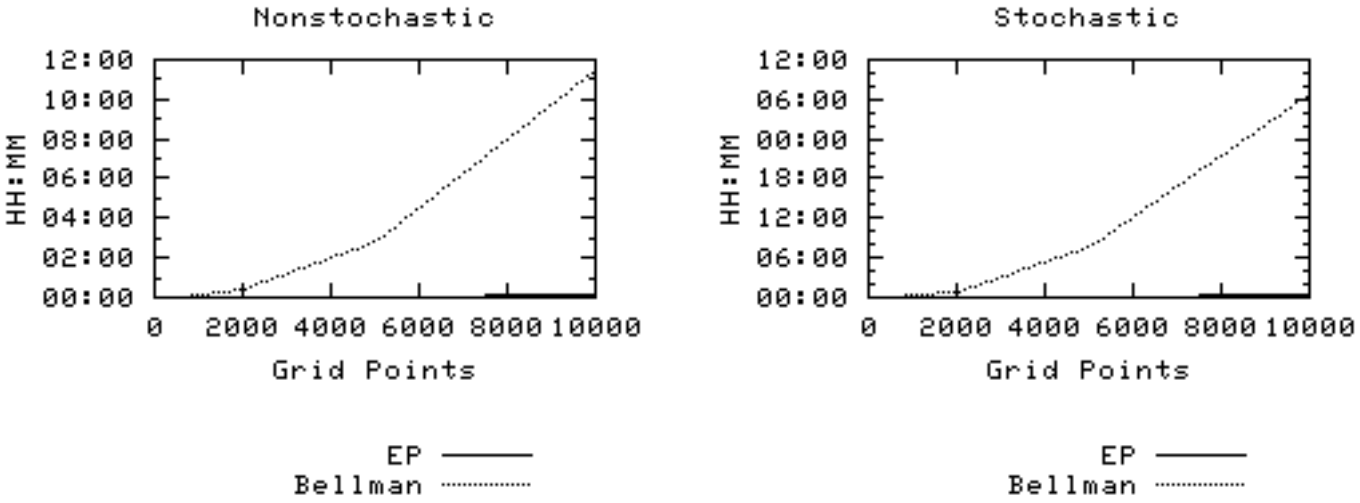
Table 3: Results for $\delta = 0.025$ (no closed form solution).

	Nonstochastic			Stochastic		
Grid Points	Evolutionary Program	Error	Bellman Iteration	Evolutionary Program	Error	Bellman Iteration
100	1.7	1	1.8	4.5	2	5.1
200	4.0	2	8.1	8.9	3	20.4
500	11.7	2	1:02.1	30.3	6	2:35.3
1,000	28.8	3	4:42.7	1:04.2	0	13:31.4
2,000	1:07.3	2	21:05.0	2:30.6	2	56:47.5
5,000	3:23.1	3	2:47:48.4	7:50.7	0	7:38:30.7
10,000	7:44.6	3	11:31:33.9	17:26.7	1	30:51:47.5

Notes: Reported CPU time is the `user` time reported by the Unix `time` command on a SPARCstation 20 with a 100 MHz HyperSPARC chip. 'Error' is the number of grid points at which the evolutionary program and Bellman equation iteration differ.

Also of interest is the case for which a closed-form solution is not available since this is the situation which typically confronts the researcher. Table 3 summarizes the results for this case (see Fig. 2 for a graphical presentation). *Qualitatively*, the same message emerges: for a large number of grid points, the evolutionary program clearly dominates in terms of CPU time. *Quantitatively*, the differences are even larger than before. In the stochastic case with 10,000 capital stock grid points, the evolutionary program finishes in less than 18 minutes while Bellman equation iteration takes over 30 hours – over 100 times longer. Both algorithms give nearly the same decision rules for capital accumulation: the

Figure 2: CPU time for $\delta = 0.025$ (no closed form solution).



maximum number of grid points which differ is 6 (for the stochastic case with 500 capital stock grid points). For a particular grid point, the two algorithms never differed by more than one grid point.

3.2. Labor–Leisure Choice

There are two reasons to be interested in this case. First, endogenous labor supply decisions are important for generating business cycle moments in the real business cycle literature. Second, the evolutionary program can be given a further workout by requiring that it solve for labor as well.[3]

The representative agent's problem in this case is:

$$\max_{\{c_t, n_t, k_{t+1}\}} \mathrm{E}_0 \left\{ \sum_{t=1}^{\infty} \beta^t [\omega \ln c_t + (1-\omega) \ln(1-n_t)] \right\}, \qquad 0 < \beta, \omega < 1 \tag{19}$$

subject to

$$c_t + k_{t+1} = z_t k_t^{\alpha} n_t^{1-\alpha} + (1-\delta) k_t, \qquad 0 < \delta, \alpha < 1, \qquad t = 0, 1, \ldots \tag{20}$$

where, in addition to the earlier variables, n_t is the fraction of time spent working. When $\delta = 1$, the decision rules are:

$$k_{t+1} = \alpha\beta z_t k_t^{\alpha} n_t^{1-\alpha}, \tag{21a}$$

$$c_t = (1-\alpha\beta) z_t k_t^{\alpha} n_t^{1-\alpha}, \tag{21b}$$

[3] An alternative, used in Bellman equation iteration, is to use an Euler equation to solve for labor supply.

and

$$n_t = \frac{\omega(1-\alpha)(1-\alpha\beta)}{\omega(1-\alpha)(1-\alpha\beta)+1-\omega}. \tag{21c}$$

The parameter values are the same as before, with the addition that $\omega = 0.33$. In implementing Bellman equation iteration, the solution for time spent working, n_t, is computed using a one dimensional nonlinear equation solver which works on the Euler equation,

$$(1-\alpha)z_t k_t^{\alpha} n_t^{-\alpha}\left(\frac{\omega}{c_t}\right) = \frac{1-\omega}{1-n_t}, \tag{22}$$

where c_t is computed from (20). This step is computationally costly, but need only be performed once for each of the $NK \times NK \times NZ$ possible configurations (next period's capital stock, this period's capital stock and this period's technology shock).

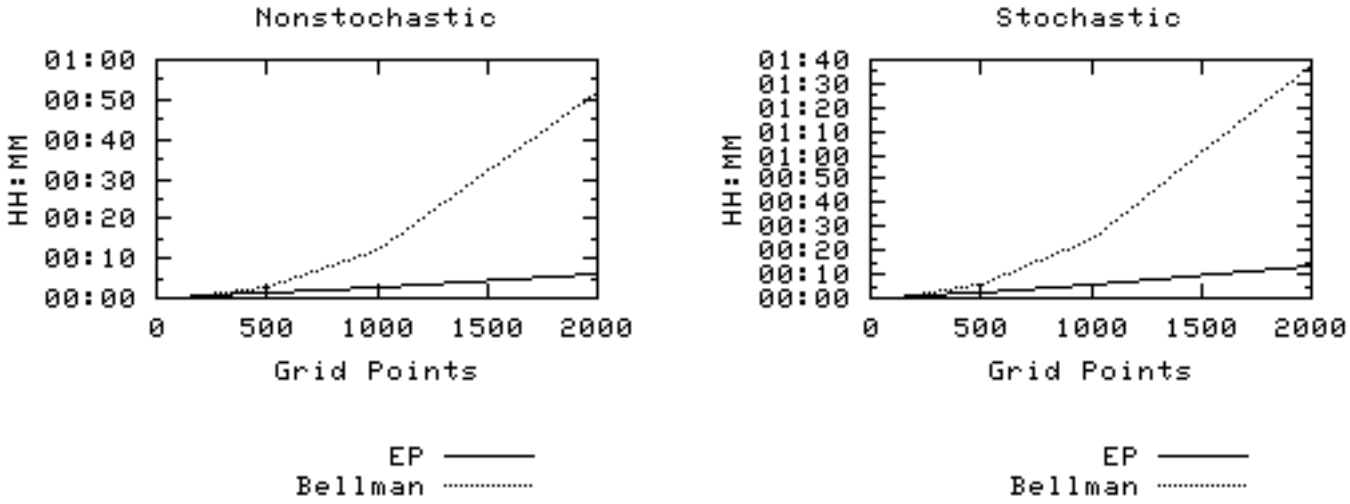
Rather than use (22) to solve for labor, the evolutionary program is required to solve not only for the capital accumulation decision but also labor decision supply. This should serve to bias the results against the evolutionary program. σ_K will now be used to control the amount of experimentation over the capital grid while σ_N will control the experimentation with respect to the labor decision. As above, the initial value for σ_K is $NK/10$ while σ_N starts at 0.1. The same convergence criteria are used as above. FORTRAN code to solve this model is reproduced in Appendix A.

Table 4: Results for the closed-form case with endogenous labor supply.

	Nonstochastic		Stochastic	
Grid Points	Evolutionary Program	Bellman Iteration	Evolutionary Program	Bellman Iteration
100	8.3	6.3	16.3	13.4
200	22.2	26.3	43.0	56.1
500	1:05.7	2:52.7	2:19.8	6:00.2
1,000	2:34.1	11:54.8	5:24.6	24:16.8
2,000	5:58.3	51:36.3	13:00.0	1:37:59.8

Notes: In all cases, the solutions were within one grid point of the known solutions given in (21a) and (21c). Reported CPU time is the `user` time reported by the Unix `time` command on a SPARCstation 20 with a 100 MHz HyperSPARC chip. The maximum error on the labor supply calculation was less than 0.01%, with this figure decreasing with the number of capital stock grid points.

Figure 3: CPU time for closed form case with endogenous labor supply.



Results for the closed-form case are presented in Table 4. Qualitatively, the results are similar to before. For a small number of grid points, there is little difference between the algorithms, with Bellman equation iteration typically completing in less time. However, for a large number of grid points, the evolutionary program performs substantially better than Bellman equation iteration. With no labor–leisure choice, the evolutionary program was about 5 times faster than Bellman equation iteration for 2,000 grid points (see Table 2). With a labor–leisure choice, the evolutionary program is over 8 times faster. These are not differences of seconds, but rather of hours. Larger capital stock grids were not attempted in this case due to the CPU and memory requirements for Bellman equation iteration.[4] The earlier results suggest that for larger grid points, the CPU time advantage of the evolutionary program would be substantial.

Finally, Table 5 summarizes the results for the case in which no closed form solution is available. Compared to the case with inelastic labor supply, there is now a greater tendency for the two algorithms to differ with respect to the capital decision rule. However, the two algorithms are always within one grid point of each other. Compared to Table 4, Bellman equation iteration takes over twice as much CPU time while the evolutionary program actually takes slightly less. At 2,000 capital stock grid points, Bellman equation iteration takes over 20 times more CPU time. Again, the differences are minutes versus hours.

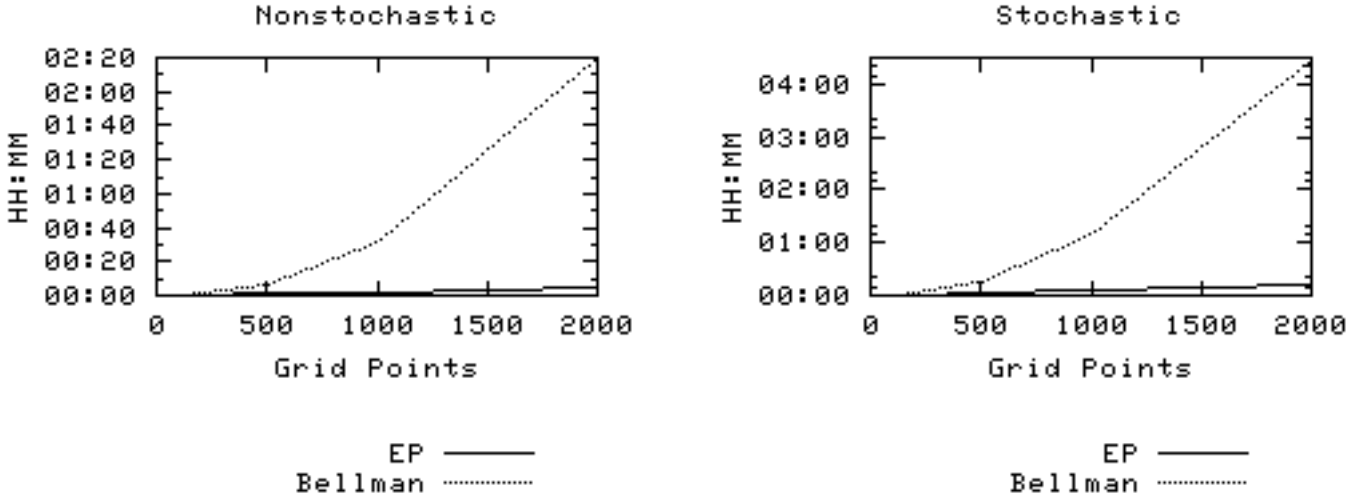
[4] Memory requirements increase since the labor supply decision is stored in memory to speed Bellman equation iteration.

Table 5: Results for $\delta = 0.025$ (no closed form is available).

	Nonstochastic			Stochastic		
Grid Points	Evolutionary Program	Error	Bellman Iteration	Evolutionary Program	Error	Bellman Iteration
100	7.0	4	12.8	16.1	9	30.5
200	17.5	2	1:00.5	37.3	9	2:06.1
500	1:01.1	15	6:38.5	1:53.2	15	16:11.1
1,000	2:18.6	18	31:01.0	4:50.5	26	1:09:09.4
2,000	5:22.9	27	2:19:53.7	11:15.6	50	4:26:55.6

Notes: Reported CPU time is the `user` time reported by the Unix `time` command on a SPARCstation 20 with a 100 MHz HyperSPARC chip. 'Error' is the number of grid points at which the capital decision rules differ for the two algorithms. Excluding these grid points, the maximum percentage difference of the labor supply decision is less than 0.01%, with this difference declining with the number of grid points.

Figure 4: CPU time for $\delta = 0.025$ (no closed form solution).



4. Conclusion

This paper described how to implement an evolutionary program to solve the Bellman equation problem for the neoclassical growth model. A total of eight cases were considered: with and without endogenous labor supply, constant versus stochastic technology shocks, and when a closed form is or is not available. When closed form solutions are available, both the evolutionary program and Bellman equation iteration deliver decision rules for

capital which are within one grid point of the known solution. When closed form solutions are not available. the evolutionary program and Bellman equation iteration produce decision rules which are quite close to each other. The most striking difference is in CPU requirements: where the evolutionary program takes tens of minutes, Bellman equation iteration takes hours. A stochastic technology shock substantially increases CPU time for Bellman equation iteration but has little effect on the time required for the evolutionary program. There is nothing in the evolutionary program algorithm which takes advantage of the fact that the large state space is due to an increase in the number of grid points for the endogenous state variable (capital). Thus, the results for, 10,000 grid points for capital should closely approximate those which would be obtained with two endogenous state variables, each with 100 grid points. This would be prohibitively expensive in terms of CPU time for Bellman equation iteration, but can be solved in a relatively short time using the evolutionary program

The neoclassical growth model was not the ultimate target for this exercise; there are many solution algorithms for this model which are even faster.[5] The neoclassical growth model provides a benchmark to evaluate the accuracy of the algorithm. Useful applications will be ones for which these other algorithms cannot be used. One such class of problem is when constraints are not necessarily binding. For example, in the neoclassical growth model one might impose a nonnegativity constraint on investment.[6] This can be handled above by making the current return to violating this constraint an arbitrarily large negative number. Another example would be a cash-in-advance economy in which money growth is at times sufficiently low that households wish to hold more money than is necessary to satisfy their cash-in-advance constraint.

[5] See, for example, King, Plosser and Rebelo (1987) and Hansen and Prescott (1995).
[6] See Christiano and Fischer (1994).

References

Arifovic, Jasmina [1995]. "Genetic Algorithms and Inflationary Economies," *Journal of Monetary Economics*, Volume 36, pp. 219–243.

Arifovic, Jasmina [1996]. "The Behavior of the Exchange Rate in the Genetic Algorithm and Experimental Economics," *Journal of Political Economy*, Volume 104, pp. 510–541.

Christiano, Lawrence J. and Jonas D.M. Fisher [1994]. "Algorithms for Solving Dynamic Models with Occasionally Binding Constraints," Minneapolis: Federal Reserve Bank of Minneapolis, Research Department Staff Report 171.

Fogel, D.B. [1992]. "Evolving Artificial Intelligence," Doctoral Dissertation, University of California, San Diego.

Hansen, Gary D. and Edward C. Prescott [1995]. "Recursive Methods for Computing Equilibria of Business Cycle Models," in *Frontiers of Business Cycle Research*, ed. Thomas F. Cooley, Princeton, New Jersey: Princeton University Press, pp. 39–54.

King, Robert G., Charles I. Plosser, and Sergio T. Rebelo [1988]. "Production, Growth and Business Cycles: I The Basic Neoclassical Model," *Journal of Monetary Economics*, Volume 21, pp. 195–232.

Prescott, Edward C. [1986]. "Theory Ahead of Business Cycle Measurement," *Federal Reserve Bank of Minneapolis Quarterly Review*, Volume 10, pp. 9–22.

Appendix A: FORTRAN Source Code

```
      program EP4
      integer NPOP, NK, NPOP2, NZ
      parameter (NPOP=20, NK=10000, NZ=2)
      integer ktemp(NK,NZ,NPOP), ik, ipop, converge, count,
     $     idx(NPOP), kold(NK,NZ), cc, krule(NK,NZ,NPOP), iz, iiz
      double precision alpha, beta, kstock(NK),
     $     vstar(NK,NZ), ktrue(NK,NZ), temp, cons, util, delta, sdk,
     $     v(NK,NZ,NPOP), ev(NPOP), z(NZ), rho(NZ,NZ), Evstar(NK,NZ),
     $     sdn, ntrue, nrule(NK,NZ,NPOP), omega, ntemp(NK,NZ,NPOP),
     $     RANMAR, GASDEV
      external RANMAR, GASDEV
C
C     Initialize random number generator and parameters.
C
      call RMARIN(28460,12031)
      alpha = 0.36d0
      beta = 0.99d0
      delta = 1d0
      omega = 0.33d0
      NPOP2=NPOP/2
      temp = omega*(1d0-alpha)/(1d0-alpha*beta)
      ntrue = temp / (temp + 1d0 - omega)
C
C     Set up grids for the technology shock and capital stock.
C
      z(1)=-0.00763d0
      z(2)=-z(1)
      z(1) = EXP(z(1))
      z(2) = EXP(z(2))
      temp = ((1d0/beta - 1d0 + delta)/(alpha*ntrue**(1d0-alpha)))
     $     **(1d0/(alpha-1d0))
      call LINSPACE(NK,kstock,0.25d0*temp,2d0*temp)
C
C     Set up rho which governs the persistence in the technology shock.
C
      rho(1,1) = 0.975d0
      rho(1,2) = 1d0-rho(1,1)
      rho(2,2) = rho(1,1)
      rho(2,1) = rho(1,2)
C
C     Initialize the decision rules.
C
      do 1000 ik=1,NK
      do 1000 iz=1,NZ
         ktrue(ik,iz) = alpha*beta*z(iz)*kstock(ik)**alpha
     $        *ntrue**(1d0-alpha)
         vstar(ik,iz) = 0d0
         Evstar(ik,iz) = 0d0
         kold(ik,iz) = 0
         do 1100 ipop=1,NPOP
            krule(ik,iz,ipop) = 1
```

```
            nrule(ik,iz,ipop) = 0.24d0
 1100    continue
 1000 continue
      do 1500 ipop=1,NPOP
         idx(ipop) = ipop
 1500 continue
      sdn = 0.05d0
      sdk = DBLE(NK)*0.1d0
      do 2000 while (sdk .gt. 0.1d0)
         count = 0
         converge = 1
C
C     Two convergence criteria: Have the decision rules for capital
C     changed recently (converge)?  Has the algorithm spent "long
C     enough" with this degree of experimentation?
C
         do 2999 while ((converge .lt. 20) .and. (count .lt. 50))
            count = count+1
            do 2100 ipop=1,NPOP
               ev(ipop) = 0d0
               do 2110 ik=1,NK
               do 2110 iz=1,NZ
C
C     Copy decision rules.
C
                  ktemp(ik,iz,ipop) = krule(ik,iz,ipop)
                  ntemp(ik,iz,ipop) = nrule(ik,iz,ipop)
C
C     For the worst half of the population, replace with a corresponding
C     member of the best half, then peturb.  There is a 50-50 chance
C     of peturbing the capital decision rule, and so a 50-50 chance of
C     peturbing the labour supply rule.
C
C     The capital decision rule is, in fact, only changed with
C     probability p.  When changed, the rule can either go up or
C     down by some random amount which is linear in the increment.
C
C     The labour decision rule is peturbed using a Normal random
C     number generator with the standard deviation being adjusted
C     downward over time.
C
                  if (ipop .le. NPOP2) then
                     ktemp(ik,iz,ipop) = MAX(MIN(krule(ik,iz,ipop+NPOP2)
     $                    + INT(GASDEV()*sdk),NK),1)
                     ntemp(ik,iz,ipop) =
     $                    MAX(MIN(nrule(ik,iz,ipop+NPOP2)
     $                    + GASDEV()*sdn,1d0),0d0)
                  endif
 2110          continue
 2100       continue
            do 2200 ipop=1,NPOP
               do 2210 ik=1,NK
               do 2210 iz=1,NZ
```

```
C
C     Compute consumption, current period utility, the value of the
C     current member at each grid point, and the "average" value
C     of the current member.
C
                  cons = z(iz)*kstock(ik)**alpha*ntemp(ik,iz,ipop)
     $                 **(1d0-alpha) - kstock(ktemp(ik,iz,ipop))
     $                 + (1d0-delta)*kstock(ik)
                  if ((cons .gt. 0d0) .and.
     $                 (ntemp(ik,iz,ipop) .lt. 1d0)) then
                     util = omega*log(cons)
     $                    + (1d0-omega)*log(1d0-ntemp(ik,iz,ipop))
                  else
                     util = -1d10
                  endif
                  v(ik,iz,ipop) = util + Evstar(ktemp(ik,iz,ipop),iz)
                  ev(ipop) = ev(ipop) + v(ik,iz,ipop)
 2210          continue
 2200       continue
C
C     Do a bubble sort of the "average" value of each member. Actually,
C     keep track of an index to the "average" values rather than copy
C     the decision rules back and forth; this step is performed (once)
C     in the loop which follows.
C
            call BSORT(ev, idx, NPOP)
            do 2300 ipop=1,NPOP
               do 2310 ik=1,NK
               do 2310 iz=1,NZ
                  krule(ik,iz,ipop) = ktemp(ik,iz,idx(ipop))
                  nrule(ik,iz,ipop) = ntemp(ik,iz,idx(ipop))
 2310          continue
 2300       continue
C
C     vstar is the BEST v across members of the population.  It speeds
C     the algorithm to keep track of the decision rule which implement
C     vstar at each grid point (point in the state space).  This rule
C     is stored in the place of the worst member in the top half of
C     the population (i.e., the part which is kept).
C
C     Two notes.
C
C     (1) The computation of vstar is in the spirit of value function
C     iteration except that rather than take a maximum over all possible
C     values of next period capital, the maximum is over all members of
C     the population.
C
C     (2) There seems to be little harm in saving the decision rule
C     which attains the maximum over members of the population at each
C     grid point since it will be discarded in the next round if this
C     turns out to be a bad decision rule.
C
            do 2400 ik=1,NK
```

```
            do 2400 iz=1,NZ
               vstar(ik,iz) = v(ik,iz,1)
               krule(ik,iz,NPOP2+1) = ktemp(ik,iz,1)
               nrule(ik,iz,NPOP2+1) = ntemp(ik,iz,1)
               do 2410 ipop=2,NPOP
                  if (v(ik,iz,ipop) .gt. vstar(ik,iz)) then
                     vstar(ik,iz) = v(ik,iz,ipop)
                     krule(ik,iz,NPOP2+1) = ktemp(ik,iz,ipop)
                     nrule(ik,iz,NPOP2+1) = ntemp(ik,iz,ipop)
                  endif
 2410          continue
 2400       continue
C
C     Do some calculations to check for convergence.
C
            cc = 0
            do 2500 ik=1,NK
            do 2500 iz=1,NZ
               cc = cc + abs(kold(ik,iz)-krule(ik,iz,NPOP))
               kold(ik,iz) = krule(ik,iz,NPOP)
               Evstar(ik,iz) = 0d0
               do 2510 iiz=1,NZ
                  Evstar(ik,iz) = Evstar(ik,iz) +
     $                 beta*rho(iz,iiz)*vstar(ik,iiz)
 2510          continue
 2500       continue
            if (cc .eq. 0) then
               converge = converge + 1
            else
               converge = 0
            endif
 2999    continue
         write(6,*) count, sdk, sdn, converge
         sdn = sdn / 2d0
         sdk = sdk / 2d0
 2000 continue
      open(unit=55, file='ep4-10000.dat', status='unknown')
      temp = -1d0
      do 9000 ik=1,NK
         write(55,10) kstock(ik), ktrue(ik,1), kstock(krule(ik,1,NPOP)),
     $        ktrue(ik,2), kstock(krule(ik,2,NPOP)),
     $        ntrue, nrule(ik,1,NPOP), nrule(ik,2,NPOP)
         temp = MAX(temp,ktrue(ik,1)-kstock(krule(ik,1,NPOP)))
         temp = MAX(temp,ktrue(ik,2)-kstock(krule(ik,2,NPOP)))
 9000 continue
      close(55)
      write(6,*) temp/(kstock(2)-kstock(1))
 10   format(8(1x,f20.10))
      stop
      end
C=======================================================================
C     A bubble sort routine.  This sorts the "average" value of members
C     of the population, but does so on an index to these members
```

```
C     rather than copying decision rules back and forth several times.
      subroutine BSORT(value, index, NN)
      integer NN, index(NN), p, swap, temp
      double precision value(NN)
 1    swap = 0
      do 1000 p=1,NN-1
         if (value(index(p)) .gt. value(index(p+1))) then
            swap = 1
            temp = index(p+1)
            index(p+1) = index(p)
            index(p) = temp
         endif
 1000 continue
      if (swap .gt. 0) goto 1
      return
      end
C=======================================================================
C     Used to initialize various grids.  Generates a series of evenly
C     spaced grid points between start and end.
      subroutine LINSPACE(N, series, start, end)
      integer N, i
      double precision series(N), start, end
      do 1000 i=1,N
         series(i) = start + (end-start)*DBLE(i-1)/DBLE(N-1)
 1000 continue
      return
      end
C=======================================================================
C     Generator of Normally distributed random numbers which mean 0
C     and standard deviation of 1.
      double precision function GASDEV()
      implicit complex (a-z)
      integer iset
      double precision v1, v2, r, fac, gset, RANMAR
      external RANMAR
      data iset /0/
      if (iset .eq. 0) then
 1       v1 = 2d0 * RANMAR() - 1d0
         v2 = 2d0 * RANMAR() - 1d0
         r = v1**2 + v2**2
         if (r .ge. 1d0) goto 1
         fac = sqrt(-2d0*log(r)/r)
         gset = v1*fac
         GASDEV = v2*fac
         iset = 1
      else
         GASDEV = gset
         iset = 0
      endif
      return
      end
```